Enabling Reconstruction of Attacks on Users via Efficient Browsing Snapshots

Phani Vadrevu*, Jienan Liu*, Bo Li*, Babak Rahbarinia†, Kyu Hyung Lee*, and Roberto Perdisci*
* Department of Computer Science, University of Georgia, USA
† Department of Computer Science, Auburn University in Montgomery, Alabama, USA
{vadrevu,jienan,boli,kyuhlee,perdisci}@cs.uga.edu, babak@aum.edu

***Abstract*—In this paper, we present *ChromePic*, a web browser equipped with a novel *forensic engine* that aims to greatly enhance the browser's logging capabilities. ChromePic's main goal is to enable a fine-grained post-mortem *reconstruction and trace-back of web attacks* without incurring the high overhead of record-and-replay systems. In particular, we aim to *enable the reconstruction of attacks that target users and have a significant visual component*, such as social engineering and phishing attacks. To this end, ChromePic records a detailed snapshot of the state of a web page, including a screenshot of how the page is rendered and a "deep" DOM snapshot, at every significant interaction between the user and the page. If an attack is later suspected, these fine-grained logs can be used to reconstruct the attack and trace back the sequence of steps the user followed to reach the attack page.**

We develop ChromePic by implementing several careful modifications and optimizations to the Chromium code base, to minimize overhead and make *always-on logging* practical. We then demonstrate that ChromePic can successfully capture and aid the reconstruction of attacks on users. Our evaluation includes the analysis of an *in-the-wild* social engineering download attack on Android, a phishing attack, and two different clickjacking attacks, as well as a user study aimed at accurately measuring the overhead introduced by our forensic engine. The experimental results show that browsing snapshots can be logged very efficiently, making the logging events practically unnoticeable to users.

I. Introduction

Web browsers have unfortunately become the preferred entry point for a large variety of attacks. For example, through the browser, a user may be exposed to malware infections via social engineering attacks [29] or drive-by downloads [13], phishing attacks [9], cross-site scripting [45], cross-site request forgery [4], clickjacking [14], etc.

While the mechanics of these attacks (i.e., how they are typically executed within the browser) are well understood, it is often challenging to determine how users arrived to a given attack page in the first place. At the same time, tracing back the steps through which an attack unfolds can be critical to fully recover from an intrusion [17] and prevent future compromises. For instance, security analysts and forensic investigators often try not only to understand how a specific attack instance was executed (e.g., find the URL from which malware was downloaded), but also attempt to put the attack into context by reconstructing the steps that preceded it [28] (e.g., whether the user fell for a social engineering attack and inadvertently triggered the malware download). While existing browser and system logs may assist in reconstructing a partial picture of how an attack page was reached, these logs are often sparse and do not provide sufficient details to precisely reconstruct the events preceding the user landing on the attack page, and what exactly happened afterwards.

Quoting [20], "we tend to lack detailed information [about an attack] just when we need it the most." Therefore, to enable a detailed *reconstruction and trace-back* of web attacks we need enhanced logging capabilities [18], [20], [26]. For instance, systems such as ClickMiner [27] and WebWitness [28] rely on full network traffic logs (or traces) and deep packet inspection to reconstruct the sequence of pages visited by users before they reach an attack page (e.g., a malware download URL). However, even by using full traffic traces, these systems are sometimes unable to precisely reconstruct all steps that brought a user to encounter an attack page, due to the complexity of modern web technologies and the consequent discrepancies between system events (e.g., user-browser interactions) and the network traffic they generate [27], [28]. Furthermore, encrypted (e.g., HTTPS) traffic would all but prevent these systems from inferring the complete path to the attack page. Other approaches, such as ReVirt [11] and WebCapsule [26], go beyond network traffic logging and analysis, and instead focus on recording fine-grained details at the system level to enable full attack replay. However, whole-system record-and-replay [10], [11] is computationally expensive, and is especially difficult to deploy on resource constrained mobile devices. On the other hand, while in-browser record-and-replay [26] can be more easily ported to mobile devices, it is hindered by difficulties introduced by OS-level non-determinism (e.g., due to thread scheduling) and can result in an inaccurate replay of browsing sessions [26], thus preventing reliable attack reconstruction. Furthermore, to enable replay, record-and-replay systems typically require storing large amounts of information, including all network traffic generated by the browser.

In this paper, we present *ChromePic*, a web browser equipped with a novel forensic engine aimed at greatly enhancing Chome's logging capabilities. ChromePic's main goal is to enable a fine-grained *reconstruction and trace-back of web attacks* without incurring the high overhead typically as-

NDSS '16, 21-24 February 2016, San Diego, CA, USA
Copyright 2016 Internet Society, ISBN 1-891562-41-X
http://dx.doi.org/10.14722/ndss.2016.23xxx

sociated with record-and-replay systems such as [11], [26]. In particular, we aim to *enable the reconstruction of attacks that target users and have a significant visual component*, such as social engineering and phishing attacks. To this end, we focus on instrumenting Google Chromium [37] (the open source project on which the Chrome browser is based) to efficiently record a browsing snapshot at every meaningful interaction between the user and the browser. For example, every time the user clicks on a page or presses a key (e.g., `Enter`), we record the input information (e.g., mouse coordinates, key code, etc.) and the page URL shown in the browser bar. Furthermore, we take a screenshot of the rendered page, and a "deep" snapshot of the related DOM tree and embedded resources (e.g., `iframes`, images, etc.). We refer to this type of detailed browsing snapshots as *webshots*. Intuitively, such rich logs would allow a security team or forensic analyst to *travel back in time* and effectively reconstruct a user's browsing actions over a desired time window. In fact, we can consider the screenshot contained in each webshot as a "video frame." These screenshots can then be stitched back together to reconstruct precisely what the user saw during every significant interaction with the browser. Furthermore, each screenshot is associated with the related full state of the DOM (including embedded objects and JavaScript source code) recorded at the very same instant in time. Namely, we record exactly how a specific page DOM was structured, how it was rendered at the time of a user-browser interaction, and how the user interacted with it, thus enabling an analysis of how the attack was implemented. Our ChromePic browser addresses the following challenges:

- *Forensic Rigor*: Our main goal (and challenge) is to enable webshots to be taken *synchronously* with user-browser interactions. Namely, let $u(t_0)$ represent a user-browser interaction u (e.g., a mouse click or key press) that occurs at time t_0. Because we aim to prevent any (potentially malicious) JavaScript code that listens on u from altering the page before the webshot is taken, our goal is to "freeze" the processing of u until we take both a screenshot of the page currently displayed by the browser as well as a full DOM snapshot. Only after the snapshot is completed the event u will be released and processed by the browser. The need for this *synchronous snapshots constraint* is motivated by the fact that we intend to prevent any discrepancy between what is logged in the webshot and what the user saw (and the DOM of the page he/she interacted with) at the very instant of time t_0 when the event u occurred.
- *Efficiency*: As attacks cannot be easily predicted, ChromePic aims to be *always-on* and continuously log webshots. This allows us to record undetected (and unexpected) attacks, in accordance with the *compromised recording* design principle [31]. However, to make *always-on* logging practical, efficiency is critical and logging overhead must be reduced to a minimum. In particular, because webshots are taken synchronously with each user input, we effectively introduce a processing overhead that increases the natural processing of input events. Therefore, the challenge we face is to make sure that no negative effect (e.g., latency) will be perceived by the user. Based on previous studies on human-computer interaction [44], we target a logging time budget of around 150ms, which would make the logging events practically unnoticeable to users. To this end, we implement a number of careful system-level browser modifications and optimizations, which we describe in detail in Section IV.
- *Transparency*: We require webshots to be taken in a transparent way w.r.t. the web pages that are being logged. For instance, there should be no easy way for (malicious) javascript code running on a page to detect whether the interactions (inputs) between the user and the page are being logged or not. In addition, webshots should also be transparent to users, in that once they are enabled the user should not notice any difference in the behavior of the browser when webshots are being recorded.

In Sections IV and V we discuss why the existing snapshot-taking capabilities currently implemented by Chromium do not satisfy the above requirements. For instance, we describe the implementation of a browser extension that attempts to meet the same requirements described above using the existing extension API, and demonstrate why such a solution is not viable.

The reader may notice that because ChromePic continuously records detailed information about the state of the browser, including visual screenshots, our system may produce numerous logs, some of which may include sensitive information. While protecting the security and privacy of the logs recorded by ChromePic is outside the scope of this paper, it is important to notice that existing solutions could be used to mitigate these concerns. For example, sensitive URL whitelisting and log encryption using a key escrow as proposed in [26] could also be used in ChromePic. We discuss these solutions in more details in Section VIII. Also, while a typical browsing session may result in numerous webshots, often the changes to a page between two consecutive user-browser interactions are minimal, thus resulting in few changes between snapshots. This provides an opportunity for storing only the *difference* between snapshots. In addition, the visual screenshots can be reduced in size using lossy compression, and the overall storage requirements for the logs of each browsing session could be further reduced using standard archiving tools. We discuss storage requirements in more details in Section VII.

In summary, we make the following contributions:

- We propose ChromePic, a web browser equipped with a novel forensic engine that aims to enable the reconstruction and trace-back of web browser attacks, especially for *attacks that directly target users and have a significant visual component*, such as social engineering and phishing.
- We develop ChromePic by implementing careful modifications and optimizations to the Chromium code base, to minimize overhead and make *always-on logging* practical. In addition, we discuss why implementing ChromePic using existing facilities, such as Chrome's extension API, is not a viable option.
- We demonstrate that ChromePic can successfully capture and aid the reconstruction of attacks on users. Specifically, we report the analysis of an *in-the-wild* social engineering download attack on Android, a phishing attack, and two different clickjacking attacks.

- We evaluate the efficiency of our solution via a user study involving 22 different users who produced more than 16.5 hours of browsing activities on hundreds of websites. We provide precise measurements about the overhead introduced by ChromePic on multiple devices, including desktop and laptop Linux systems as well as Android tablet devices. Our results show that the vast majority of webshots can be taken very efficiently, making them practically unnoticeable to users.

II. WebShots

As mentioned in Section I, we aim to enable the reconstruction and trace-back of web attacks that target users, with particular focus on attacks that have a significant visual component, such as social engineering and phishing attacks. To this end, we design ChromePic to embed an *always-on forensic engine*. Specifically, we instrument the Chromium browser to record rich logs, called *webshots*, that aim to capture the state of the rendered web pages at every significant interaction between the user and the browser.

A. What is a WebShot?

A *webshot* consists of the following components: (i) a timestamp and other available details about the user input event that triggered the webshot (e.g., mouse event type and related screen coordinates, keypress code, etc.); (ii) the full URL of the page with which the user interacted; (iii) a screenshot of the currently visible portion of the web page (the *viewport*) rendered by the browser; (iv) a "deep" DOM snapshot that consist of the page's DOM structure, all embedded objects (e.g., the content of all images), the DOM and embedded objects of all `iframe`s, the JavaScript code running on the page, etc.

To satisfy the forensic rigor requirement mentioned in Section I, webshots must be taken *synchronously* with the triggering user input. This requirement, along with the *always-on* operational goal for our ChromePic system, has a significant impact on the amount of overhead we can afford for producing each webshot. In Section IV, we describe a set of very careful code instrumentations and optimizations that make efficient webshots feasible.

B. Input Events that Trigger a WebShot

WebShots are triggered by user interactions with web pages. In theory, we could take a screenshot for every single "raw" user input event, including every mouse movement, every key-down event, every tap or gesture on a touchscreen, etc. However, many user input events (e.g., most mouse movements) have no real changing effect on the underlying web page. Furthermore, to reduce overhead, it is desirable to minimize the type and number of events that actually trigger a webshot. At the same time, our goal is to capture enough webshots to allow for the reconstruction and trace-back of possible attacks. To balance these conflicting goals, we trigger a webshot for each of the following events:

- *Mouse Down*: Mouse clicks are a common interaction between users and web pages. Clicks often have important consequences, such as initiating the navigation to a new page, submitting a form, selecting a page element, etc. As each click starts with a *mouse down* event, we trigger a webshot for each such event.
- *Tap*: On touchscreen devices, taps are the initial event for a variety of gestures, including "clicking" on a link or button. Therefore, a tap often (though not always) has an effect similar to a mouse down event. Therefore, we trigger a webshot at every tap event.
- `Enter` *Keypress*: In many cases, pressing `Enter` has the same effect as a mouse click, such as submitting a form, navigating to a new link, etc. Therefore, we trigger a webshot at every *keydown* event for the `Enter` key.
- *Special Keys*: We also trigger a screenshot every time a special key is pressed. For example, pressing `tab` while entering data in a form usually allows to transition from an input field to another, thus indicating that the previous field has been fully entered. Other keys (e.g., the space bar) may be used to scroll a page or pause/start a video, or to navigate to the previous page (e.g., using the `backspace`). We have selected a total of five special keys whose *raw keydown* event triggers a webshot.
- *Generic Input Events*: All other input events, such as mouse movements, mouse wheel, key presses, etc., that do not fall within the above categories are also considered. Specifically, we trigger a webshot for each "generic" event, but impose a time constraint: if the previous webshot has been taken more than a predefined number of seconds ago (e.g., 5 seconds), we take another webshot, otherwise we skip this event (i.e., no webshot is taken). Notice that this time constraint only applies to "generic" input events, and to the case when a key is kept pressed. For all other single events mentioned earlier (e.g., mouse down, tap, etc.) we always take a webshot, regardless of the time.

Webshots are logged synchronously with the triggering input event, as required by the *forensic rigor* property introduced in Section I. Effectively, the user input will be held from processing until a full webshot is taken. In Section IV, we will explain that because user inputs are processed on the render thread of Chromium's renderer process, this has the effect of preventing the DOM of the page from changing before the webshot is recorded. Hence, each webshot reflects what the user saw at the moment of her interaction with the page. This has the effect of preventing attempts from the attacker to hide the attack by altering the log, for example by rapidly changing the DOM and appearance of the page immediately after a user-browser interaction.

III. Use Cases

In this section, we discuss a representative use case scenario, to highlight how our system could be used in practice. In general, we envision ChromePic to be particularly useful in aiding the *reconstruction and trace-back* of attacks that involve user actions, such as social engineering and phishing attacks. In these cases, reconstructing what the user saw or what exact information was entered on a phishing page is critical to understand how the attack unfolded. We argue that *these types of attack are difficult to reconstruct without a visual account of what the user experienced.* ChromePic would ideally be deployed in corporate and government network environments, where web-based attacks may represent the first step of larger incidents (e.g., targeted attacks). At the same time, we believe ChromePic may also be useful in other scenarios, such as in web application debugging.

Example Use Case: Meet Bob, a corporate employee who, while using the browser at work, falls victim to a social engineering malware download attack [29] by clicking on a misleading advertisement. Once installed, the malware opens a backdoor to the corporate network, which is later used by the malware owners to gain access to and exfiltrate sensitive information, triggering a data breach detection (e.g., due to side effects such as selling of information in the underground markets). Then, a forensic analysis team is hired to investigate how the data was leaked. By analyzing network logs, such as web proxy logs that report all URLs visited by the corporate network users, the forensic analysts notice something anomalous (e.g., a particularly suspicious set of URLs) in Bob's web logs recorded a week earlier. Therefore, the analysts ask for authorization to explore Bob's *ChromePic logs*. Finally, *by exploring the webshots produced by our system*, the analysts are able to reconstruct the social engineering attack that tricked Bob into installing the initial malicious software.

By learning how Bob fell for the attack, including obtaining a precise reconstruction of the visual tricks used for the social engineering attack, the corporate network security team could then develop a user training session on social engineering, to better educate corporate employees on how to decrease the likelihood of becoming a victim [15]. In addition, by having both the screenshot taken at the very moment when the user clicked on the misleading ad, as well as the related full DOM snapshot, this information could be used to enhance browser-based defenses against social engineering [3].

Notice that ChromePic enables the reconstruction not only of the exact moment in which the attack is triggered (e.g., a click on a social engineering malware ad), but also of the sequence of pages with which the user interacted before falling for the attack. In addition, in case of phishing attacks ChromePic would also provide an account of the exact information the user leaked on the phishing site. Knowing what information was "phished" may be important to decide what actions to take to mitigate the damage to both the user and to the corporate network (e.g., the user may have leaked access credentials related to sensitive corporate assets).

IV. System Details

A. Background

Before we present the details of ChromePic, we first provide a brief overview on the Chromium browser architecture. As Chromium's architecture is fairly complex, we will limit the following description to highlight only those components that are needed to understand how our code modifications and optimizations work.

Chromium uses a multi-process architecture [37], which includes a main browser process, called *Browser*, and one rendering process, called *Renderer*, per each open browser tab[1]. The Browser runs multiple threads [42], including a *UI Thread*, which handles UI events among other things, and an *IO Thread*, which handles the IPC communications [36] between the Browser and all Renderers. Each Renderer is also multithreaded [35]. The Renderer's *Main Thread* is responsible for communicating via IPC with the Browser, whereas the Renderer's *Render Thread* is responsible for rendering web content, including executing JavaScript code.

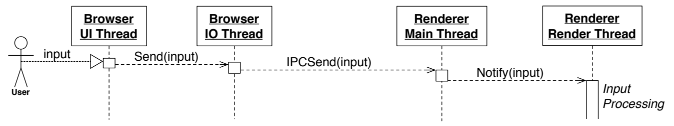


Fig. 1. Overview of how user inputs to a web page are passed to the Renderer Thread. Dashed arrows indicate asynchronous calls. Notice that the function names are intentionally simplified, and do not exactly reflect the (long chains of) function calls that exists in the source code.

As shown in Figure 1, user inputs to a web page are first received by the Browser's UI Thread, and then asynchronously communicated via IPC (by the IO Thread) to the Renderer that is responsible for the tab where the page is rendered. The IPC message will first be processed by the Renderer's Main Thread, and then forwarded to the Renderer's Render Thread [35]. For instance, a click on a hyperlink will be processed by the Render Thread, to decide wether a navigation event should be triggered. Furthermore, JavaScript code execution (e.g., initiated due to a *listener* registered on the input event), is also executed in the context of the Render Thread.

B. ChromePic Overview

Figure 2 provides a simplified overview of how our ChromePic browser generates a webshot. Notice that all dashed arrows in the figure represent *asynchronous* calls.

In response to a user input, ChromePic takes the following main actions: (1) on the Browser process, it calls Chromium's code for taking a screenshot of the current visible tab (see details in Section IV-D), to which the user input is destined; (2) it opens a file that will be used to save the DOM snapshot and passes its file descriptor, `fd`, to the Renderer, along with the user input; (3) as the input and `fd` are received by the Renderer, it saves the current entire DOM, including embedded objects and JS code, in MHTML format; (4) once the DOM snapshot has been saved, the Renderer waits for confirmation from the Browser that the screenshot taking process has terminated; only then, (5) the user input is processed using the original Renderer's workflow. In this process, notice that if the screenshot finishes before the DOM snapshot is saved, there will simply be no delay between steps (3) and (5).

The high-level steps described above allow us to guarantee that each webshot is taken *synchronously* with the user input, and no DOM modification due to the current input is allowed before the webshot is logged, in accordance with the forensic rigor requirement stated in Section I. Moreover, our webshot events are *transparent* to the (possibly malicious)

[1]In practice, a Renderer may in some cases be responsible for rendering more than one tab [37]. To simplify our description, in the following we will assume one tab per Renderer. Also, we will not consider out-of-process-iframes [38], which are a recent ongoing project.

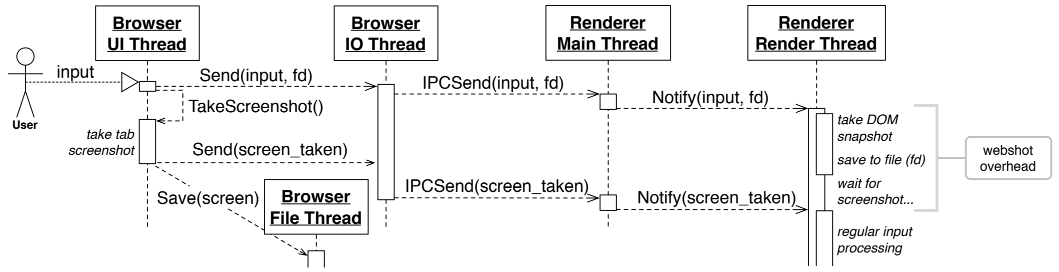


Fig. 2. Simplified view of how ChromePic processes user inputs that trigger a webshot. Dashed arrows indicate asynchronous calls.

page. ChromePic's code is designed so that after logging the input can continue its "natural" processing path, and no information regarding the webshot events is transferred to the page (notice that while side-channel attacks cannot be excluded, user input timings are not easily predictable, thus making detecting the existence of ChromePic a laborious, non-deterministic endeavor).

Challenges. While the process of taking synchronous screenshots may appear straightforward at first, our design of ChromePic faces two main challenges. First, we had to spend countless hours to learn the intricacies of the enormous Chromium code base. The limited documentation for many of the modules we instrumented forced us to a great deal of "reverse engineering" of the source code. In fact, our code modifications had to span not only multiple processes, but also multiple threads per process (UI, IO, Renderer, GPU, etc.). In addition, while we strived to limit the number of changes to existing code as much as possible, to meet our *efficiency* requirements we had to engineer a number of optimizations, so to minimize the webshot overhead shown in Figure 2.

C. Identifying the Target Renderer Process

Every Renderer Process has a corresponding `RenderProcessHost` object in the Browser process, which is used to send and receive IPC messages between the two processes. Effectively, the `RenderProcessHost` represents the Browser side of a single Browser-Renderer IPC connection [37]. A `RendererProcessHost` object communicates with multiple `RenderWidgetHost` instances, each one representing one tab in the browser [35]. For every `RenderWidgetHost` object, we create a custom `SnapshotHandler` object whose responsibility is to coordinate the process of taking webshots for a given tab. When an input event is received, the responsible `RenderWidgetHost` object is identified by the Browser, and represents the last point in the Browser after which the event is passed on to the correct Renderer via IPC message. We take control of the input event just before it is passed on to the Renderer, and handle the event via our `SnapshotHandler` instead. By doing so, we are able to identify the correct `RoutingID` for the IPC messages [36], and therefore we can coordinate the process of taking a snapshot with the appropriate Render Thread.

D. Taking Screenshots Efficiently

One way to implement the `TakeScreenshot` function shown in Figure 2, would be to call Chromium's `CopyFromCompositingSurface` and simply wait for the `CopyFromCompositingSurfaceFinished` callback (see Figure 3). However, we empirically found that this process sometimes takes a large amount of time to finish (e.g., several hundred milliseconds, depending on the web page). Obviously, a large latency would be unsustainable for our purposes, as it violates our efficiency requirements. Therefore, we had to break down and study the details of the process used by Chromium to satisfy `CopyFromCompositingSurface`. While documentation such as [34], [39] helped, this was not an easy task, as it required a much deeper understanding of the internals of Chromium's compositing process than found in the sparse Chromium project documents.

We then discovered that to efficiently take synchronous screenshots we could safely use the process depicted in Figure 3. Specifically, to satisfy `CopyFromCompositingSurface`, the Browser relies on a graphics library (GL) API and assistance from the GPU (with code running on the GPU process, or GPU thread in Android [34]). The GL/GPU module in Figure 3 is represented separately from the Browser UI thread for presentation convenience (to be more precise, the `DrawFrame` and `GetFrameBufferPixels` functions are actually executed asynchronously within the context of the Browser's UI thread. Only the *ReadBack* part of the screenshot taking process is executed on the GPU process/thread).

In simplified terms, we can break down the screenshot-taking process into five main steps: (1) draw (i.e., composite the layers of) the web page; (2) copy the pixels; (3) crop/scale; (4) read back the final bitmap; (5) save to file (we execute the file saving process within the Browser's File Thread [42]). However, we found that once step (2) is completed, the screenshot has effectively been taken, and do not need to wait for the crop/scale operation before we can "release" the user input for further processing. Namely, after step (2) the screenshot content is not going to be influenced by the processing of the input, even if the input causes the DOM to change.

The `DrawFrame` operation is controlled by the compositor scheduler `cc::scheduler`, which takes into account factors such as the device's v-sync and dynamically establishes a target rate at which frames are drawn [33]. For instance, on devices with a v-sync frequency of 60Hz, a frame would be ideally drawn every ~16ms. Thanks to these properties, the time between the arrival of the user input and our `Send(screen_taken)` message in Figure 3 is typically on the order of only few tens of milliseconds (see Section VII).

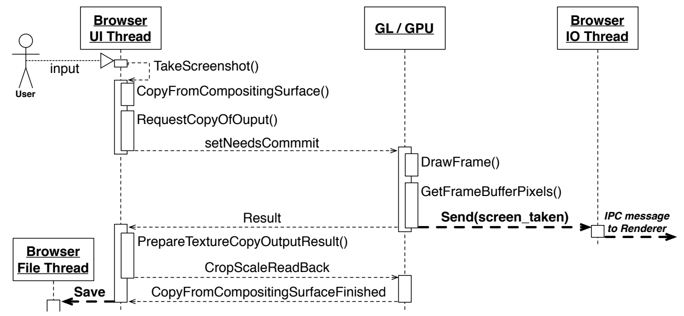


Fig. 3. Overview of how screenshots are taken and the Renderer is notified (notice that some of the function call names have been shortened and made more readable for presentation purposes, compared to the source code).

E. Taking "Deep" DOM Snapshots Efficiently

Along with each screenshot, we also take a "deep" DOM snapshot that not only includes the current structure of the DOM (at the time of the input), but also the content of all frames, embedded objects (e.g., images), and javascript code. To enable these rich DOM snapshots, we apply several important changes to Chromium's code for saving web pages in MHTML format [30]. Specifically, we enhance Chromium's code to include javascript code into the DOM snapshots and, importantly, to significantly improve efficiency. Below, we focus on detailing these latter code optimizations.

To save a page in MHTML format, Chromium implements a `GenerateMHTML` function, which can be called in the Browser process from the UI Thread. Given a specific tab, this function is responsible for *serializing* the tab's web page content into MHTML format, and to save it into a file. However, the Browser does not have direct access to the DOM of the page in each tab. Therefore, to save the MHTML content the Browser must rely on the Renderer process. But because the Renderer executes within a sandbox, it cannot directly open a file to save the MHTML content. Chromium's solution is to (1) open a file in the Browser process; (2) pass the file descriptor of the already opened file to the Renderer; and (3) ask the Renderer (via IPC message) to produce the MHTML content of the main page and each frame it embeds, and to save it into this file.

Unfortunately, instead of sending only one IPC message to the Renderer for the entire process, Chromium's code results into sending one IPC message to request to save the main page, as well as one separate IPC message to the Renderer to request the saving of each frame embedded in the page[2]. Because modern complex web pages contain a potentially large number of frames (e.g., an iframe for each ad embedded in the page), the full process of serializing and saving the MHTML content can be quite expensive, lasting from hundreds of milliseconds to a few seconds. Therefore, using the existing code to take a DOM snapshot synchronously with each user input would violate our efficiency requirements.

One of the reasons why the above process is highly inefficient is that for each IPC that is received, the Renderer creates a `Task`, which will (asynchronously) run on the Render Thread, at a time decided by the Renderer Scheduler [32]. As mentioned in [33], "the render thread is a pretty scary place," due to its complexity. Its execution "routinely stalls for tens to hundreds of milliseconds [...] on ARM, stalls can be seconds long" [33]. The reason is that there are many different types of tasks that share the same Render Thread processing time. For example, execution commonly stalls due to the execution of "long" javascript code [41]. Therefore, each task related to a frame's MHTML seralization could easily find itself starving for CPU time, thus bloating the overall time needed to complete the full DOM snapshot.

To address the above challenges and dramatically reduce the overhead related to taking DOM snapshots, we use the following approach. Instead of calling `GenerateMHTML`, thus generating multiple IPC messages to the Renderer specifically dedicated to MHTML serialization, we send only one IPC message to the Renderer. In fact, we piggyback the "take DOM snapshot" message from the Browser onto the input-passing IPC message that the Browser already must send to the Renderer to communicate the user input (see Figure 2). To this end, we modify the `IPCSend(input)` IPC message to also carry a file descriptor parameter, `fd`, which is related to a file we explicitly open to allow the Renderer to save the DOM snapshot. In addition, we instrument the input-processing task that would normally only process the user input, so that when its `Task` is executed it will first serialize the page into MHTML format, and then simply continue with the regular user input processing, as shown on the right side of Figure 2. This guarantees that the DOM snapshot is taken synchronously with the related user input event, and before any input-driven DOM changes can occur.

There is one remaining question: how can we serialize both the main page and all embedded frames, considering that the original `GenerateMHTML` code needed to send multiple IPC messages? To solve this problem, we write new MHTML serialization code to explicitly traverse the entire frame tree from the Render Thread, sequentially serialize each frame, and save the entire DOM snapshot into the file previously opened by the Browser (in Section VIII we discuss how this process could be further adapted in the future, once out-of-process-iframes [38] become enabled by default).

V. Alternative Implementation using Extensions

In this section, we discuss whether webshots could be captured using Chrome's extension API [7]. An extension-based implementation would be appealing, because it does not require any browser instrumentation and can easily be added to existing browser releases. However, we will show that an extension-based implementation of ChromePic is not a viable alternative in practice, due to the constraints imposed by the browser's extension API itself and to the higher average overhead associated with webshots produced via the extension. In the following, we refer to the extension-based version of ChromePic as ChromePicExt.

A. ChromePicExt Overview

Because our *forensic rigor* requirement (see Section I) dictates that webshots must be taken synchronously with the user input, the `content javascript` component of

[2]This approach will be useful in the future, once the out-of-process-iframes project is completed and the functionality is turned on by default, as discussed later in the paper.

ChromePicExt must intercept user inputs before any page javascript code. This goal can be achieved in two steps: (1) by setting the `run_at` property in the extension's manifest file to `document_start`; and (2) by registering an event listener for user input events (e.g., `mousedown`, `keydown`, etc.) on the `window` object as soon as the `content javascript` starts being executed. All ChromePicExt's event listeners are registered with the `useCapture` option set to `true`, to guarantee that the `content javascript` will be the first to capture and handle the event, before any page javascript has a chance to receive the same event. In the following, we will use `cnt.js` to refer to the extension's `content javascript`.

During the interaction between the user and the browser, if a particular event that ChromePicExt is listening on is fired, our listener captures it first and passes the event object to the handler function implemented by the extension. At this point, ChromePicExt's `cnt.js` needs to temporarily stop the propagation of the event object to any other listeners, including listeners registered by the web page with which the user is interacting, until a full browsing snapshot is taken. This is crucial to ensure that the other event listeners will not have an opportunity to change the appearance of the web page before the screenshot and DOM snapshot are recorded. Notice also that because the execution of javascript code within each renderer process is single-threaded[3] [40], the input event cannot be processed by any other listener until `cnt.js` "yields." Furthermore, because `cnt.js` runs in an isolated world[4], the page javascript cannot observe or interfere with the extension's event processing.

Once a triggering event is received, to take a screenshot of the rendered content `cnt.js` needs to call the `captureVisibleTab` API accessible via the `background` extension component. At the same time, `cnt.js` also needs to produce a snapshot of the current page's DOM tree, which could be achieved for example by asking the `background` component to call the `pageCapture.saveAsMHTML` API. Once both the screenshot and DOM snapshot are taken, the event can be released so that the browser can propagate it to other listeners.

Challenges. In Chrome, the `content javascript` component of an extension has limited direct access to the extension API. The full extension API can be accessed via the `background` component. In the following, we refer to the `background` extension component as `bgnd.js`, for short. The `cnt.js` and `bgnd.js` components can communicate via message passing. For instance, immediately after a user input is captured, `cnt.js` can use `sendMessage` and ask `bgnd.js` to call `captureVisibleTab`, thus producing a screenshot of the current page with which the user is interacting. Unfortunately, the simple approach described above does not satisfy the *forensic rigor* requirement for webshots, because `bgnd.js` runs in the extension process [6], rather than the renderer process where `cnt.js` runs, and the screenshot is therefore taken asynchronously. Similarly, to take a full DOM snapshot `bgnd.js` can make use of `saveAsMHTML`, but this also causes the DOM snapshot to be taken asynchronously w.r.t. the user input. In other words, if the `cnt.js` simply captures a user event, asks `bgnd.js` to take a webshot (using `captureVisibleTab` and `saveAsMHTML`) and then immediately "yields," the event can be propagated by the browser to other listeners. Therefore, there is no guarantee that the page javascript will not change the page (DOM and rendering) before the webshot is actually logged.

[3]Notice that WebWorkers cannot directly change the DOM.

[4]https://developer.chrome.com/extensions/content_scripts

Possible Solutions. One possible approach to make the webshot taking functionality synchronous may be for `cnt.js` to actively wait (e.g., loop) until `bgnd.js` communicates that the webshot request has been processed via a callback function. However, because JavaScript execution within each process is single-threaded[5], this would prevent `cnt.js` from yielding to the callback function, because it would need to run in the renderer process where `cnt.js` is actively waiting. Therefore, this would stall the renderer process and thus the web page (we have empirically verified all observations). Another solution could be to "sleep," instead of actively waiting, for example by leveraging `setTimeout` or `setInterval`. However, this would not solve the problem, because while `cnt.js` "sleeps," it effectively "yields" and the captured user event will trickle down to the next listeners, thus again violating the requirement that webshots must be taken synchronously.

One may think that `cnt.js` could simply capture an event object, say e, and (1) make a deep copy of the object, thus creating $e' = e$; (2) cancel the propagation of e to the remaining listeners[6]; (3) wait until the callback from `bgnd.js` indicates that the webshot has been taken; and (4) re-dispatch the event by injecting e', so that the browser will propagate the user event to the remaining listeners. Unfortunately, this will cause the `isTrusted` property of e' to be set to `false`, thus potentially preventing some listeners from correctly processing the event. In addition, the value of `isTrusted` would allow an attack page to infer the presence of ChromePicExt, and perhaps stop the attack to prevent it from being logged/analyzed, thus violating the *transparency* requirement (see Section I).

B. Our Approach

To solve the above problems, we proceed as follows. First, we will focus only on how to synchronously take a screenshot, and then discuss how to take a DOM snapshot.

Once a message has been sent to `bgnd.js` to ask for a screenshot to be taken, `cnt.js` actively waits for the screenshot to be completed. However, as mentioned earlier, `cnt.js` cannot actively wait for a callback from `bgnd.js`, as this would bring `cnt.js` to stall. Instead, what `cnt.js` can do is: (1) explicitly choose the name of the file where the screenshot should be stored; (2) pass this information to `bgnd.js` (via sendMessage) and at the same time ask it to concretely start the screenshot capturing process; (3) actively probe the file system using a *synchronous* XMLHttpRequest to the local URL [7] to test whether the screenshot file has been saved (via `captureVisibleTab`).

[5]WebWorkers cannot be used in the scenario we are considering.

[6]Using `Event.stopPropagation()`

[7]Notice that this can be enabled in the extension's manifest file, via the `web_accessible_resources` parameter.

```
// Save "shallow" DOM snapshot
var domSnapshot = document.head.outerHTML + document.body.outerHTML;
chrome.runtime.sendMessage(command: "save_dom", dom: domSnapshot);

// Take screenshot
var md_time = Date.now();
var filename = "snapshots/"+md_time+".png";
var xhr_request = new XMLHttpRequest();

chrome.runtime.sendMessage(command: "take_screenshot", file: filename);
while(true) {
  try {
    xhr_request.open('GET', chrome.extension.getURL(filename), false);
    xhr_request.send(null); // send synchronous request
    break;
  } catch (err) {
     // Synchronous XMLHttpRequest has failed
  }
}
```

Fig. 4. Simplified `cnt.js` source code.

Figure 4 shows a simplified code snippet that implements the approach outlined above. The synchronous XMLHttpRequest will raise an exception if the file does not exist. In this case, `cnt.js` will try again, until the file can be found on disk (or a maximum number of attempts have been exhausted, as a safeguard to avoid waiting indefinitely in case of failure at the extension process side). After `cnt.js` exits the active wait loop, the user input event will effectively be "released" and passed by the browser to the remaining listeners, thus allowing the processing of the event to continue (e.g., this could trigger some DOM modification by the underlying page javascript).

Unfortunately, the approach described above cannot be used to synchronously take a DOM snapshot using `saveAsMHTML`. The reason is that while the call to `saveAsMHTML` happens asynchronously via `bgnd.js`, which runs within the extension process, ultimately `saveAsMHTML` will delegate the responsibility of producing and saving the `mhtml` representation of the DOM to the same Renderer process where `cnt.js` also runs, within the Render Thread (see Section IV-E). Therefore, if `cnt.js` actively waits for the `mhtml` file to be saved it will simply wait indefinitely, as the `mhtml` file cannot be produced until `cnt.js` "releases control" of execution on the renderer's main thread. One way to avoid this problem is to simply program `cnt.js` to save the DOM structure, as shown at the top of Figure 4. However, this is a much more limited, "shallow" representation of the DOM, compared to what can be obtained with `saveAsMHTML`, because embedded objects (e.g., the content of images or iframes) are not saved.

The `cnt.js` could be extended to produce a result that is more similar to `saveAsMHTML`. For instance, the content of images can be accessed by first loading them into a `canvas` and then reading the content of the canvas [25]. But this is a quite cumbersome and inefficient operation. Also, while cumbersome, it would be possible to communicate (e.g., via `postMessage`) to the `cnt.js` running in the context of the embedded frames[8] to produce a DOM snapshot, which could then be combined to the DOM of the main page to produce a more comprehensive, "deep" snapshot of the page.

It should be apparent by now that the extension-based approach to taking synchronous webshots is sort of a "hack," in that it bypasses some of the restrictions imposed by the browser on `cnt.js` and its inability to directly access the extension APIs. Furthermore, screenshots cannot be made fully transparent to the user, because every time a screenshot is taken the browser visually indicates that a file is being downloaded (on the bottom of the browser window). In Section VII, we will also show that the extension-based implementation of ChromePic imposes a higher overhead, compared to the browser instrumentation approach described in Section IV. Overall, this demonstrates that extensions are not suitable for meeting all of ChromePic's design requirements.

VI. Reconstructing Attacks on Users

In this section, we report on a number of experiments that demonstrate how ChromePic can capture attacks on users, and enable their post-mortem reconstruction. Specifically, we will discuss three attacks, an *in-the-wild* social engineering download attack on Android, a phishing attack, and two clickjacking attacks proposed in [1].

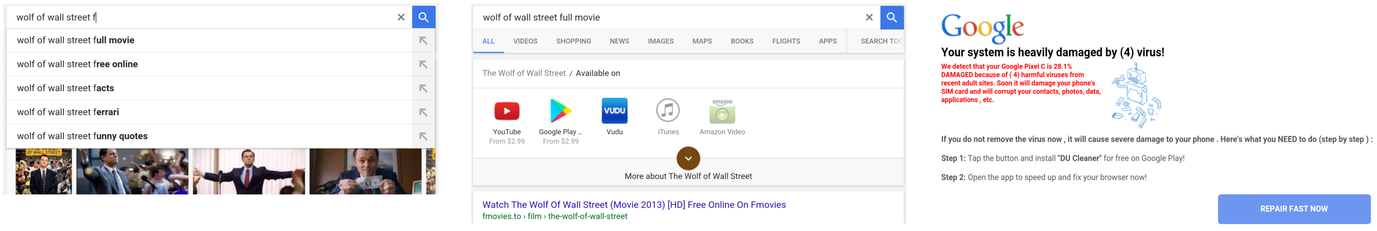
A. Social Engineering Download Attack

During our user study (see Section VII-B), we encountered an *in-the-wild* social engineering download attack. Here is how a user arrived to this attack: (1) The user visits www.google.com and searches for "wolf of wall street"; (2) after scrolling the results, the user modifies the search terms by adding the letter "f" to the search string (see Figure 5a); (3) the search engine suggests "wolf of wall street full movie" as the top search suggestion, which is clicked (with a touch screen tap) by the user; (4) the user then clicks on the top search result, which redirects the browser to a site called fmovies[.]to; (5) as the site loads, with no interaction from the user, an advertisement embedded in the page forces the browser to open a new tab where a page is loaded from us.intellectual-82[.]xyz; (6) an alert popup window is immediately shown, which warns the user that the device is infected by multiple viruses; (7) clicking on the OK button makes the alert window disappear, but the user now sees the us.intellectual-82[.]xyz page (which was previously in the background) claiming that the Android device is "28.1% DAMAGED because of 4 harmful viruses" (see Figure 5c) and recommends the user to download an application called "DU Cleaner"; (8) clicking on a "REPAIR FAST NOW" button, the user is redirected to the Google Play store, and specifically to information about an app called GO Speed[9] (not DU Cleaner, as stated on the attack page).

Using ChromePic, we were able to record all main steps of the attack. In fact, the screenshots in Figure 5 were all taken by ChromePic and confirm that using the recorded webshots, the social engineering attack described above can indeed be reconstructed by tracing back the user-browser interactions, including tapping on the "REPAIR FAST NOW" button on the attack page. Naturally, after the user clicks on this download button and control is passed to the Google Play app, ChromePic could not follow the next user actions (e.g., whether the app was installed or not on the device). This is expected, as ChromePic is meant to reconstruct all steps of web-based attacks that unfold within the browser. The GO Speed app that the user is asked to install seems to be benign, as it has been installed by a large user base (more than 10M users, according to Google Play) and an analysis of VirusTotal.

[8]Assuming the `all_frames` option is set to true in the manifest file.

[9]https://play.google.com/store/apps/details?id=com.gto.zero.zboost&hl=en


wolf of wall street f
wolf of wall street full movie
wolf of wall street free online
wolf of wall street facts
wolf of wall street ferrari
wolf of wall street funny quotes
wolf of wall street full movie
ALL
VIDEOS
SHOPPING
NEWS
IMAGES
MAPS
BOOKS
FLIGHTS
APPS
The Wolf of Wall Street / Available on
YouTube From $2.99
Google Play ... From $2.99
Vudu
iTunes
Amazon Video
More about The Wolf of Wall Street
REPAIR FAST NOW

Watch The Wolf Of Wall Street (Movie 2013) [HD] Free Online On Fmovies
fmovies.to › film › the-wolf-of-wall-street

Google

Your system is heavily damaged by (4) virus!

We detect that your Google Pixel C is 28.1% DAMAGED because of (4) harmful viruses from recent adult sites. Soon it will damage your phone's SIM card and will corrupt your contacts, photos, data, applications , etc.

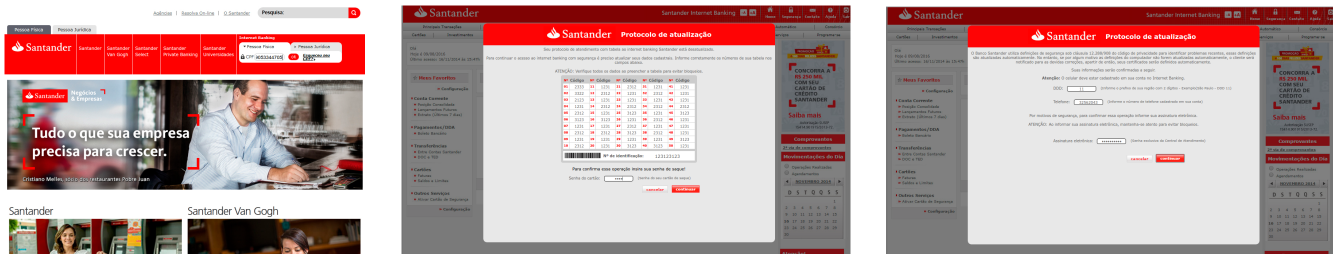
If you do not remove the virus now , it will cause severe damage to your phone . Here's what you NEED to do (step by step) :

Step 1: Tap the button and install "**DU Cleaner**" for free on Google Play!

Step 2: Open the app to speed up and fix your browser now!

(a) (b) (c)

Fig. 5. Some of the screenshots captured by ChromePic during an *in-the-wild* social engineering "fake-AV-like" attack on Android.


Santander
Tudo o que sua empresa precisa para crescer.
Santander Van Gogh
Santander
Protocolo de atualização
Santander Internet Banking

(a) (b) (c)

Fig. 6. Some of the screenshots captured by ChromePic during a phishing attack (the attack URL was first reported in PhishTank, submission #4359181).

com reports no anti-virus labels[10]. After analyzing the DOM snapshots taken by ChromePic, we suspect that the attackers are trying to monetize an advertisement campaign that pays for every new "referred" installation of the app. For instance, the attack page contains a link to click.info-apps[.]xyz and another to tracking.lenzmx[.]com with a URL query parameter `mb_campid=du_cleaner_tier2`. After a mouse click, the browser is redirected (via HTTP 302 redirections) through those two sites to the final `market://` URL referring to the GO Speed app. It is likely that the FakeAV-like advertising tactics employed in this social engineering attack are simply a way to convince more users to install the app and (illicitly) increase revenue, in a way similar to how pay-per-install networks [19], [43] monetize third-party software installations.

There is a small exception to be noted. ChromePic did not take a screenshot of the alert popup window, which should have been triggered by the user clicking on the OK button to close the popup. The reason is that alert windows are rendered "out of context" w.r.t. to browser tabs, and our current implementation of ChromePic does not support taking a snapshot when users interact with such alert windows (we plan to add support for alert windows in future releases of ChromePic). However, it is worth noting that by analyzing the DOM snapshots taken as the user interacted with the attack page (at us.intellectual-82[.]xyz) we were able to also reconstruct the content of the alert popup:

*WARNING ! This Google Pixel C is infected with viruses and your browser is seriously damaged. You need to remove viruses and make corrections immediately. It is necessary to remove and fix now. Don't close this window. ** If you leave , you will be at risk ***

B. Phishing Attack

Besides tracing-back the steps followed by users who reach an attack page, ChromePic can also assist in understanding how the user interacted with the attack itself. For instance, in the case of phishing attacks, our webshots capture a wealth of information about what data was leaked by the user. To demonstrate this, we present an example of a recent phishing attack posted on PhishTank (submission #4359181[11]).

After using ChromePic to visit the phishing URL, which impersonates a Brazilian banking website, we simulated the actions of a user who falls for the attack by providing fake information (due to format-checking javascript, we had to figure out how to provide fake but syntax-compliant data). Figure 6 shows some of the snapshots taken by ChromePic as we interacted with the attack website. Unlike other phishing attacks, which are often limited to stealing the victim's login credentials, this attack is fairly sophisticated as it attempts to reproduce the entire banking site. Once the user logs in (by providing his/her CPF[12] code), the site claims the balance of the user's bank account has been hidden (presumably for security purposes) and must be recovered. As the user clicks on a menu bar link, the site requires the victim to fill in a set of security codes, as shown in Figure 6b. Notice that here the attacker is attempting to essentially steal the user's entire *security code card*[13]. By doing so, the attackers will subsequently be able to perform any bank transaction operation without being blocked by the real bank's security mechanisms. Finally, after the user provides the security code card information, the phishing site also requests the user's telephone number and password (Figure 6c). Once this information is provided, the site shows a "loading" animation that makes the user believe his/her data is being verified (not shown in Figure 6 due to space constraints). But at this point the attack has already succeeded.

[10] sha1: 811b367c4901642ae41b4b8f0167eac2d3ac4039

[11] http://www.phishtank.com/phish_detail.php?phish_id=4359181

[12] https://en.wikipedia.org/wiki/Cadastro_de_Pessoas_F%C3%ADsicas

[13] An English language explanation of how security code cards are used in financial applications can be found at this link: https://www.interactivebrokers.com/en/?f=%2Fen%2Fgeneral%2FbingoHelp.php

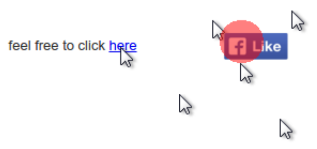


Fig. 7. *Destabilizing pointer perception* clickjacking attack.

C. ClickJacking Attacks

To demonstrate how ChromePic is able to also capture *clickjacking* attacks, we reproduced two attacks described in [1]: the *Destabilizing Pointer Perception* attack and the *Peripheral Vision* attack. The (simulated) attacks, which we adapted from publicly available code[14] by the authors of [1], are available at https://chromepic.github.io/clickjacking [15].

Destabilizing Pointer Perception: The attack is shown in Figure 7. In this attack, the user intends to click on a "here" hyperlink. However, as the mouse pointer approaches the link, the following events occur: (1) a fake pointer is drawn that has a left-side displacement error, compared to the real pointer (which is hidden); (2) as the user brings the fake pointer on top of the link, the real pointer is actually located on the Facebook `Like` button; (3) because the `Like` button is rendered within a third-party frame, the attack javascript cannot hide the mouse pointer at this time, therefore, the attack instead draws other random mouse pointers to confuse the user and effectively prevent the user from noticing that the real mouse pointer is over the `Like` button; (4) as the user attempts to click on "here," the real click actually occurs on the `Like` button, thus completing the clickjacking attack.

Figure 7 shows the screenshot taken by ChromePic at the *mouse down* event. The center of the red circle is the exact location where the user input event occurred. Notice that the fake mouse pointers are captured by the screenshot, including the pointer located over "here." At the same time, the real mouse pointer is not captured in the screenshot, because it is drawn by the OS, not rendered by the browser (only the fake pointers are rendered by the browser). Nonetheless, the coordinates of the real input event are recorded in our webshot, and it is therefore straightforward to find the correct location of where the real mouse pointer was located and draw the red circle accordingly over the screenshot. Also, by analyzing the DOM snapshots produced by ChromePic, it is easy to recover the fact that the mouse is hidden via CSS (using `cursor:none`), and to also get the full source code for the javascript functions that enable the attack, including the creation of fake mouse pointers (see Figure 8).

Peripheral Vision: In this attack, the objective is to attract the user's attention towards an area of the screen that is far from where the mouse clicks actually occur. To this end, a game is setup, as shown in Figure 9. In this game, the user needs to click on the `Play` button on the bottom left of the screen, so to catch the moving `L` or `R` blocks within the purple box on the right side of the screen. Because the user's attention is drawn

[14]http://wh0.github.io/safeclick-blast/list.html

[15]The original attack code is currently broken due to a missing remote file; after analyzing the code we found an easy fix and we were able to recreate the attacks.

```
function distract() {
    var img = document.createElement('img');
    img.className = 'random';
    img.src = 'http://i.imgur.com/EWmYMN2.png';
    img.style.top = Math.random() * 160 + 160 + 'px';
    img.style.left = Math.random() * 160 + 240 + 'px';
    playarea.appendChild(img);
    var dummy = img.clientHeight;
    img.style.top = Math.random() * 160 + 160 + 'px';
    img.style.left = Math.random() * 160 + 240 + 'px';
    setTimeout(function () {
        playarea.removeChild(img);
    }, RANDOM_MOVE_TIME);
}
```

Fig. 8. Reconstruction of code for generating fake pointers from ChromePic's DOM snapshots.

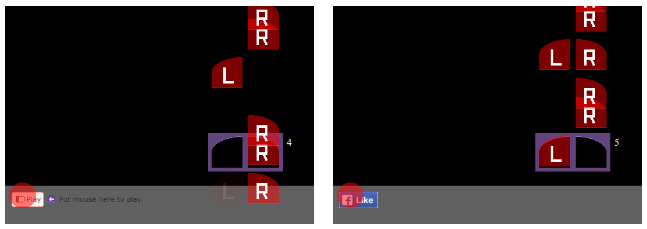


Fig. 9. Two screenshots that capture the *peripheral vision* clickjacking attack.

to the right side, while the clicks occur on the bottom left, the user may not notice that at some random convenient time the attacker may replace the `Play` button with a Facebook `Like` button. If the mouse click occurs when the `Like` button is displayed, the clickjacking attack succeeds.

Figure 9 shows two screenshots, taken at two different *mousedown* events. In the screenshot on the left, the user clicks on the real `Play` button. The screenshot on the right shows that at the second *mousedown* event the `Play` button had temporarily (for only one second) been replaced with the `Like` button, which received the user's click. As can be seen, ChromePic correctly captured the two events (the center of the red circle represents the exact location where the user clicked). Notice that this attack again has a significant visual component that would be difficult to reconstruct by analyzing only the page DOM, and that we were able to correctly capture it thanks to ChromePic's ability to take screenshots *synchronously* with the user inputs.

VII. Performance Evaluation

In this section, we present a set of experiments dedicated to measuring the overhead introduced by webshots.

A. Experimental Setup

Our ChromePic browser is built upon Chromium's codebase version 50.0.2626.2. Our source code modifications amount to approximately 2,000 lines of C++, which we plan to make available to the security research community.

We evaluate ChromePic on both Android 6.0 on a Google Pixel-C tablet with an Nvidia X1 quad-core CPU and 3GB of RAM, as well as on two machines running Linux Ubuntu 14.04: a Dell Optiplex 980 desktop machine with a quad-core Intel Core-i7 processor and 8GB of RAM, and a Dell Inspiron 15 laptop with a Core-i7 CPU and 8GB of RAM.

TABLE I. DATA COLLECTED DURING *User Study 1*

Platform	**# Users**	**Browsing time** (minutes)	**Sites visited**	**Pages visited** (webshots on)	**Pages visited** (webshots off)	**WebShot events**
Android	16	363	92	480	479	2428
Linux laptop	15	346	80	777	746	2145
Linux desktop	11	286	65	369	404	1376
Total	22 (unique)	995	204 (unique)	1626	1629	5949

B. User Study Setup

User Study 1: To evaluate the overhead imposed by our code changes to Chromium, we perform a user study involving 22 distinct users (with IRB approval). Specifically, we compile our ChromePic browser for both Linux and Android, and ask the study participants to use the devices described earlier for generic Internet browsing activities. Users were allowed to freely browse any site of their choosing. The only restriction we imposed was to avoid visiting any website containing personal data, such as online banking sites, their Facebook page, etc., to avoid recording any sensitive information. Each user was asked to perform one or more browsing sessions on different devices, with each session lasting approximately 15 minutes. Each user completed no more than two separate browsing sessions per device (a few users used only the Android and Linux laptop devices, and did not browse on the desktop Linux machine). Overall, we collected 363 minutes of browsing activity on the Android tablet from 16 different users, 346 minutes on the Linux laptop from 15 users, and 286 minutes on the Linux desktop from 11 users (more than 16.5 hours of browsing overall), which included several thousands input events per device. The users visited more than 1,600 different web pages (i.e., URLs) on 204 distinct web sites (i.e., different effective second-level domains, including google.com, youtube.com, amazon.com, and several other highly popular sites), producing close to 6,000 webshots overall. Table I reports a summary of the data we collected.

For this study, the browser was setup so that webshots are active only on randomly selected pages. Namely, every time the user navigates to a new page, the browser "flips a coin" and decides if the webshot logging capabilities should be activated or not (other experiments described later had the webshot logs always on). The reason for this is that we wanted to measure and compare the time needed by the browser to process input events with and without our code changes, to demonstrate that our webshots do not impose any other input processing delay, besides the actual time to record the logs. We comment on the results of this experiment in Section VII-C (see also Figure 11).

User Study 2: We also performed a smaller targeted user study involving 4 different users browsing on the Linux laptop device (with webshots always on). In this study, we asked the users to login into sites such as Facebook, Gmail, Twitter, Google Drive, etc., using a "temporary" account we created only for this study, which therefore contains no true personal information. This experiment aimed at evaluating ChromePic's overhead during activities such as writing emails, writing Facebook/Twitter posts, writing a GoogleDoc text document, etc. Overall, we collected 53 minutes of browsing time. The experimental results are discussed in Section VII-C.

User Study 3: Finally, we performed a separate small user study involving 6 users to evaluate the performance of ChromePicExt, the extension-based implementation that attempts to record browsing snapshots similar to the webshots recorded by ChromePic (see Section V). We discuss the related results in Section VII-C.

C. ChromePic Performance Measurements

User Study 1: In Table II, we report a breakdown of the overhead measurement results performed on browsing traces collected during our *User Study 1*. Specifically, we report the 50th percentile (i.e., the median) and 98th percentile of the time required for taking screenshots, "deep" DOM snapshots, and for the total webshots time. All numbers are in milliseconds.

To better explain how the measurements in Table II are obtained, let $u(t_0)$ be a user input event that occurs at time t_0, which triggers a webshot. Also, let t_{sn} be the time at which the `screen_taken` notification is sent in Figure 3 from the GL module to the Browser IO Thread. Namely, this is the time when the screenshot has actually been captured, and the user input can be processed (see Section IV-D). On the other hand, let t_{sc} be the time when the `CopyFromCompositingSurfaceFinished` callback is called. We define the *screenshot notification* time as $(t_{sn} - t_0)$, the *screenshot callback* time as $(t_{sc} - t_0)$.

Similarly, let t_d be the time at which the DOM snapshot has been saved, and the user input can be further processed, as discussed in Section IV-E (see also Figure 2), and δ_f be the time taken to save the DOM snapshot to file using the MHTML format. The *DOM snapshot time with file write* is computed as $(t_d - t_0)$, whereas *DOM snapshot time w/o file write* is equal to $(t_d - \delta_f - t_0)$, which therefore excludes the time needed to copy the snapshot to file. The reason why we measure this latter quantity is that with some more engineering effort the DOM snapshot file saving process could be moved to a separate Renderer process thread, thus effectively decreasing the overhead imposed by the DOM snapshot logging.

The total webshot time is computed as $(\max\{t_{sn}, t_d\} - t_0)$, according to the discussion provided in Section IV. This time could be further reduced to $(\max\{t_{sn}, (t_d - \delta_f)\} - t_0)$ by offloading the DOM file saving process to a separate Renderer process thread (we leave this implementation task to future releases of ChromePic).

Figure 10 shows the distribution of the total time needed to log the webshots on different devices, while Table II reports a breakdown of the webshot times into their components (50th- and 98th-percentiles). On both Linux devices (laptop and desktop) 98% of all webshots are logged in less than 120ms. This is a very good result, because any latency below 150ms is practically unnoticeable to users [44]. On Android, 98% of webshots are logged in less than 264ms, with a median time of around 78ms. While the 98th-percentile time is higher than our 150ms target, it is still a low overhead that is on

TABLE II. *User Study 1* - PERFORMANCE OVERHEAD (50TH- AND 98TH-PERCENTILE)

Platform	**Total with file write** (ms)	**Total w/o file write** (ms)	**Screenshot notification time** (ms)	**Screenshot callback time** (ms)	**DOM snapshot time with file write** (ms)	**DOM snapshot time w/o file write** (ms)
Android	78.05, 263.06	59.53, 203.01	13.02, 25.88	65.67, 109.97	77.55, 261.81	58.86, 202.38
Linux laptop	39.16, 118.43	33.32, 109.55	5.38, 27.68	36.17, 71.05	38.95, 118.26	33.12, 109.38
Linux desktop	22.36, 93.19	19.02, 76.11	2.74, 23.80	38.95, 118.04	22.11, 85.86	18.74, 70.53

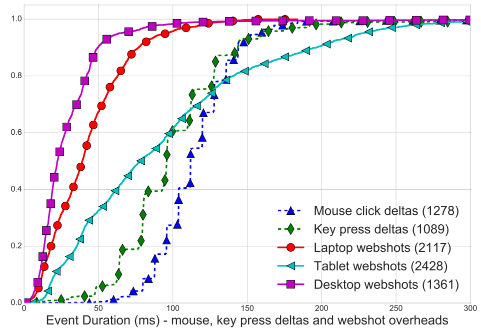


Fig. 10. Time needed to take webshots and comparison with *mouse-down/up* and *key-down/up* time deltas (the number of events on which the CDFs are computed are in parenthesis).

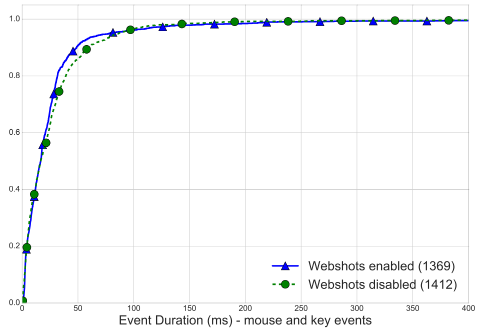


Fig. 11. Comparison of "natural" input event processing time with and without webshots enabled (the number of events on which the CDFs are computed are in parenthesis).

the very low end of the "noticeable" latency classification provided in [44]. Also, our results indicate that 82.02% of all webshots on Android can be taken in less than 150ms. Furthermore, notice that the total overhead is driven by the DOM snapshot time, including saving the DOM to file, rather than the screenshot notification time. From Table II, we can see that the 98th-percentile of the total webshot logging time for Android could be reduced to roughly 203ms if file saving was delegated to a separate thread in the Renderer process. In addition, in this setting 89.33% of the webshots on Android would take less than 150 ms.

To further put our results into perspective, we also compared the time needed to take webshots to the time in between *mouse-down/up* and *key-down/up* events. In other words, we measure the time that it takes for a user to lift her finger from the mouse button or from a key. The *mouse-down/up* and *key-down/up* time deltas are measured on the Linux desktop, with webshots turned off. As we can see from Figure 10, the distribution (CDF) of webshot overhead times on the Linux laptop and desktop are always to the left of the *mouse-down/up* and *key-down/up* time deltas curves. Because we start the webshot log at the *down* event, this means that in the vast majority of cases when a mouse click or a key press occurs, the *webshot will be fully logged by the time the user raises her finger*. Also, the Android tablet curve is almost entirely to the left of the *mouse-down/up* and *key-down/up* curves, showing that even on Android the webshots can be taken efficiently.

Another result worth noting is that our screenshot code optimizations, described in Section IV-D, yield a very significant overhead improvement, as can be seen by comparing the notification time and callback time in Table II.

To verify that our webshots do not negatively impact the subsequent "natural" input processing times, in Figure 11 we also compare the amount of time taken by the browser to process a user input in two different cases: when webshots are disabled (dashed line), and when the input is processed right after a webshot has been logged (solid line). Specifically, let t_0 be the time when the Browser sends a user input u to the Renderer, and t_i be the time when the Renderer confirms to the Browser that the input has been processed (we use Chromium's `LatencyInfo` objects to measure this). Also, let t'_i be the "input processed" confirmation time related to events that triggered a webshot, and δ_w be the time delta needed to log a webshot. The first (dashed) curve measures $(t_i - t_0)$, which represents the "natural" input processing time. Similarly, the second (solid) curve measures $(t'_i - \delta_w - t_0)$, which represents the time needed by the browser to process the input *after* a synchronous webshot has been taken (see Figure 2). As can be seen, the two curves are very similar, indicating no unexpected delay to the natural input processing time due to webshot events. In other words, the webshots do not cause any other delays, besides the actual time to take the webshots, δ_w.

User Study 2: As discussed in Section VII-B, we separately measured the overhead for user activities on popular web sites, such as Facebook, Twitter, Gmail, Google Drive, etc. We recorded thousands of user input events, 1,910 of which triggered a webshot. Of these webshots, 50% were processed in less than 66ms, and 98% took less than 240ms. Furthermore, 80% of all the webshot took less than 150ms. After closely analyzing the measurements, we found that the slight increase in overhead, compared to *User Study 1*, was due to DOM snapshots on Gmail, due to how the page is structured (e.g., Gmail pages embedded a larger number of `iframe`'s and had a larger DOM size). Specifically, the 98th-percentile for the total webshot time on Gmail was around 245ms. The times on all other popular sites (Facebook, Twitter, Google Docs, etc.) were in line or even lower than those obtained in *User Study 1*. For instance, on Facebook the 98th-percentile was less than 120ms. Overall, if we exclude Gmail from this experiment, 98% of the webshots can be taken in 108ms.

User Study 3: For comparison purposes, we also measured the performance of taking screenshots using the extension-based

TABLE III. AVERAGE STORAGE REQUIREMENTS (MBS/MINUTE)

Platform	Uncompressed		Compressed	
	Screenshots	DOM	Screenshots	DOM
Android	6.80	11.62	0.31	0.54
Linux laptop	4.66	11.33	0.15	0.88
Linux desktop	2.31	8.07	0.09	0.83

approach discussed in Section V. These have been done on the desktop machine, and the results should therefore be compared to the third row of Table II. Also, notice that in this experiment we are only considering the screenshot time (as explained in Section V, it is not easy to take synchronous "deep" DOM snapshots via the extension API). We found that 50% of screenshots require at least 140ms and 98% of them require 243ms. This is in contrast with the 2.74ms and 23.80ms, respectively, that are required by the browser-based version of ChromePic. Furthermore, the extension times are much larger than the total time needed to take a full webshot (including the DOM) on the desktop machine using the instrumented browser solution (see Table II). This reinforces our conclusion that an extension-based solution is not only cumbersome, as discussed in Section V, but also much less efficient.

D. Storage Requirements

Table III shows the storage requirements for archiving the webshots produced during *User Study 1*. After a straightforward compression process (converting screenshots to JPG and using lossless compression for DOM snapshots), the webshots take a maximum of 1.03MB per minute of browsing on the Linux laptop. Android logs required only 0.85MB/minute of storage, and 0.92MB/minute on the desktop machine. This space requirements could be further reduced by using lossy compression on the DOM-embedded images, for instance by converting them to a low- or medium-quality JPG.

Let's now consider a scenario in which ChromePic is deployed in a corporate network setting. Assume that in average users spend half of their working time (4 hours/day) browsing, while the other half is spent on other tasks (meetings, development, design, data analysis, etc.). If we assume 22 business days per month, and 1.03MB of storage needed per minute of browsing (i.e., the maximum amount we observed), a single user would produce less than 6GB of webshot logs per month. In a corporate network with 1,000 users, this would result in less than 6TB of storage for an entire month of browsing logs for the network, or 72TB for an entire year of logs. Considering that a multi-TB hard drive currently costs only a few hundreds US dollars, an entire year of webshot logs could be archived for only a few thousand US dollars. In alternative, considering that business-grade cloud-based storage services are currently priced at less than \$0.03/GB per month, archiving *one entire year worth of webshots* for the entire corporate network in the cloud would cost less than \$2,200 per month[16].

VIII. DISCUSSION

There exist some corner cases in which it is not possible to "freeze" the state of the DOM/rendering immediately after a user input arrives. For instance, if a user input arrives while the Render Thread is already executing another task, such as a long-running javascript program that affects the DOM, the processing of the user input will have to wait until javascript terminates, and until its own `Task` is scheduled for execution (see Section IV-E). The net effect is that the DOM snapshot will reflect the state of the DOM after the already running javascript code terminates. Notice, however, that this is also true for "natural" input processing. Namely, the input will apply to the modified DOM, regardless of whether a webshot is taken or not. Therefore, our snapshots correctly reflect the state of the DOM at the time when the input becomes effective. Similarly, because screenshots need to wait for the compositor to redraw, the exact instant in time in which the screenshot is taken is determined by the `cc::scheduler` (see Section IV-D). If an animation is in progress on the page, it may be possible for the screenshot to be one (or a very small number of) frame(s) "off" w.r.t. the user input. Again, this also holds for "natural" input processing (i.e., even if webshots were disabled), because the input may become effective after a redraw.

In Section IV-E, we mentioned that once the out-of-process-iframes (OOPIFs) [38] project is completed and becomes active by default, we will need to slightly adapt our code for taking DOM snapshots. In fact, we believe that OOPIFs would allow us to further decrease the time needed to take a snapshot. The reason is as follows. Assume the user interacts (e.g., clicks on a link) with a page that embeds several `iframes` (e.g., to display different ads). In the current implementation, both the main page and `iframes` are processed in the same Renderer process. Therefore, the DOM of the main page and all `iframes` has to be produced at once, synchronously with the input (see Section IV-E). But with OOPIFs we could produce all these partial DOM snapshots in parallel by simply sending a "take DOM snapshot" IPC message to the main page and all `iframes` at the same time.

Because ChromePic continuously logs user-browser interactions and takes screenshots, the recorded logs may contain sensitive user information. We argue that the solutions proposed in [26] could also be readily applied to ChromePic's output. For instance, ChromePic could employ a customizable whitelist of sites on which webshots should be turned off. To be more strict, ChromePic could be prevented from logging any events on pages loaded via HTTPS that have a valid (not self-signed) TLS certificate. Furthermore, a "helper" application (or a separate browser thread) could be responsible for continuously gathering and encrypting the browser logs. Specifically, because ChromePic can log a unique ID for each new browser tab, all logs can be easily attributed to their own specific tab. Thus, the helper application could perform the following high-level actions: (1) generate an encryption key for every new tab; (2) encrypt all logs related to a tab with that tab's key; (3) once the tab is closed (or earlier), store the key into a *key escrow* [8], along with meta-data related to the user/machine, the time when the tab was opened and closed, and the set of domain names visited within the tab; (4) "forget" the tab's key.

The key escrow could be owned by the user or, in enterprise environments, by the machine's administrator, and the keys released only when a security investigation is called for. In addition, because each tab can be stored separately and encrypted with a different key, investigators can be selectively

[16]Estimated using http://calculator.s3.amazonaws.com/index.html (with Cold HDD)

given access only to some tabs rather than the entire browsing history. The decision on whether to authorize the decryption of a tab would depend on the specific investigation, but could for instance be based on the time frame in which the attack is suspected to have happened, and on the list of domains that have been visited within the tab, which can be recorded as meta-data and encrypted with a "global" key. We leave the engineering of this key escrow-based system to future work.

IX. Related Work

The analysis of security incidents is often hindered by the lack of necessary logs. As mentioned in [20], "it is all too often the case that we tend to lack detailed information just when we need it the most." The existing logging functionalities provided by modern operating systems and browsers are often insufficient to precisely reconstruct an attack. Below we discuss previous works that aim to enhance logging and improve the ability to investigate security incidents.

Enhanced Logging. To enable the analysis of security incidents, Kornexl et al. [18] propose a network "Time Machine," whose goal is to efficiently record detailed information extracted from network traffic. The purpose of this system is to support forensic analysis and network troubleshooting. To increase efficiency and allow for storing network traffic information for long periods of time, Time Machine only records the first portion of each network connection. Even with partial recordings, [18] demonstrates that this approach enables the analysis of security incidents. Krishnan et al. [20] propose a virtualization-based forensic engine to keep track and record access to data objects read from disk. The proposed system follows the chain of access operations on the objects as they are copied into memory and accessed by different processes. The output is an audit log that enables the reconstruction of the sequence of data changes. Ma et al. [23] develop a low cost audit logging system for Windows, which aims to enable accurate attack investigation and significant log reduction.

The instrumentation of Chrome has been proposed in the past in different security contexts. For example, Bauer et al. [5] propose an information-flow tracking system that allows for enforcing fine-grained browser security policies. Excision [2], is an instrumentation of Chrome that aims at detecting and blocking the inclusion of malicious third-party content into web pages. To this end, Excision keeps track of the origin of third-party content to be loaded as part of the page.

Our ChromePic system is different, in that it ams to introduce fine-grained logging in Chromium to enable the recording and post-mortem investigation of web-based attacks, with particular focus on attacks on users that have a significant visual component.

Record-and-Repaly Systems. ReVirt's main goal is to enable whole-system record-and-reply [11]. To this end, it uses a virtualization-based approach to log detailed information about a VM's guest system execution instruction-by-instruction. This enables deterministic replay of the entire system, thus also allowing an exact replay of previously recorded intrusions. Other whole-system record-and-replay engines, such as PANDA [10], share similar goals. Whole-system record-and-replay is expensive and difficult to deploy on resource-constrained mobile devices. To obviate these problems, Neasbitt et al. propose WebCapsule [26], which aims to enable browser-level record-and-replay. WebCapsule is implemented by instrumenting Blink, Chrome's rendering engine. Because recording occurs at a higher level, compared to [11], WebCapsule does not allow for fully deterministic replay. On the other hand, WebCapsule is portable to multiple platforms, including mobile devices.

ChromePic is different from the above systems, in that it does not aim to enable replay. Rather, our system aims to introduce very low overhead, and to record enough detailed information about the state of the browser to enable an accurate reconstruction of web-based attacks towards users, especially for attacks with a significant visual component, such as social engineering and phishing.

Automated Incident Investigation. WebWitness [28] is an incident investigation system that leverages deep packet inspection to reconstruct the steps followed by users who reach social engineering or drive-by malware download pages. The system relies on full network packet traces to performs a (network-based) analysis of both the content of web pages and the way in which the content is requested (e.g., by analyzing referrers and HTTP redirections), and is able to reconstruct the web browsing path that brought the user to the final attack page. Unlike WebWitness, which is purely based on an analysis of network traces using a set of heuristics and inference methods, ClickMiner [27] aims to reconstruct the path to an attack page by replaying network traces into an instrumented browser.

BackTracker [17] is a system for automatically reconstructing the sequence of steps followed by an attacker to compromise a machine. Given an initial detection point, such as a malicious file identified by a security analyst, BackTracker traces back processes and files that have a causal relation to the detection point, by leveraging OS-level logs. The final result is a dependency graph that explains what system objects affected (or caused) the presence of the malicious file on disk, thus potentially revealing the attacker's entry point into the system. Taser [12] and RETRO [16] use OS-level logs and perform forward tracking to identify and recover form intrusions, whereas other recent works [21], [22], [24] have focused on improving accuracy in backward- and forward-tracking of intrusions, and on reducing the space for OS logs.

Our work is different from the systems discussed above, in that ChromePic's main goal is to produce highly efficient fine-grained browser logs that could be used to enable and improve the accuracy on automated incident investigation systems.

X. Conclusion

In this paper, we presented *ChromePic*, a web browser equipped with a novel *forensic engine* whose goal is to greatly enhance the browser's logging capabilities. ChromePic enables a fine-grained post-mortem *reconstruction and trace-back of web attacks* without incurring the high overhead of record-and-replay systems. ChromePic works by recording a detailed snapshot of the state of a web page, including a screenshot of how the page is rendered and a "deep" DOM snapshot, at every significant interaction between the user and web pages. If an attack is later suspected, these fine-grained logs can be processed to reconstruct the attack and trace back the sequence of steps the user followed to reach the attack page.

We developed ChromePic by implementing several careful modifications and optimizations to the Chromium code base, to minimize overhead and make *always-on logging* practical. Using both real-world and simulated web attacks, we demonstrated that ChromePic can successfully capture and aid the reconstruction of attacks on users. Our evaluation included the analysis of an *in-the-wild* social engineering download attack on Android, a phishing attack, and two different clickjacking attacks, as well as a user study aimed at accurately measuring the overhead introduced by our forensic engine. The experimental results showed that browsing snapshots can be logged very efficiently, making snapshot logging events practically unnoticeable to users.

References

[1] D. Akhawe, W. He, Z. Li, R. Moazzezi, and D. Song, “Clickjacking revisited: A perceptual view of ui security,” in *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, Aug. 2014.

[2] S. Arshad, A. Kharraz, and W. Robertson, “Include me out: In-browser detection of malicious third-party content inclusions,” in *Proceedings of the 20th International Conference on Financial Cryptography and Data Security (FC)*, 2 2016.

[3] L. Ballard, “No more deceptive download buttons,” 2016, https://security.googleblog.com/2016/02/no-more-deceptive-download-buttons.html.

[4] A. Barth, C. Jackson, and J. C. Mitchell, “Robust defenses for cross-site request forgery,” in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS ’08, 2008.

[5] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian, “Run-time monitoring and formal analysis of information flows in Chromium,” in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, Feb. 2015.

[6] Chrome, “Background pages,” https://developer.chrome.com/extensions/background_pages.

[7] ——, “Extensions,” https://developer.chrome.com/extensions.

[8] D. E. Denning and D. K. Branstad, “A taxonomy for key escrow encryption systems,” *Commun. ACM*, vol. 39, no. 3, pp. 34–40, Mar. 1996.

[9] R. Dhamija, J. D. Tygar, and M. Hearst, “Why phishing works,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI ’06, 2006.

[10] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, “Repeatable reverse engineering with panda,” in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, ser. PPREW-5, 2015.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, “Revirt: Enabling intrusion analysis through virtual-machine logging and replay,” *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, Dec. 2002.

[12] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara, “The taser intrusion recovery system,” in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP ’05, 2005.

[13] C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, N. Provos, M. Z. Rafique, M. A. Rajab, C. Rossow, K. Thomas, V. Paxson, S. Savage, and G. M. Voelker, “Manufacturing compromise: The emergence of exploit-as-a-service,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS ’12, 2012.

[14] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson, “Clickjacking: Attacks and defenses,” in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[15] S. Institute, “A multi-level defense against social engineering,” 2003, https://www.sans.org/reading-room/whitepapers/engineering/multi-level-defense-social-engineering-920.

[16] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, “Intrusion recovery using selective re-execution,” in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’10, 2010.

[17] S. T. King and P. M. Chen, “Backtracking intrusions,” in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’03, 2003.

[18] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann, and R. Sommer, “Building a time machine for efficient recording and retrieval of high-volume network traffic,” in *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC ’05, 2005.

[19] P. Kotzias, L. Bilge, and J. Caballero, “Measuring pup prevalence and pup distribution through pay-per-install services,” in *25th USENIX Security Symposium (USENIX Security 16)*, Aug. 2016.

[20] S. Krishnan, K. Z. Snow, and F. Monrose, “Trail of bytes: Efficient support for forensic analysis,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS ’10, 2010.

[21] K. H. Lee, X. Zhang, and D. Xu, “High accuracy attack provenance via binary-based execution partition,” in *NDSS*, 2013.

[22] ——, “Loggc: garbage collecting audit log,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security*, ser. CCS ’13, 2013.

[23] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu, “Accurate, low cost and instrumentation-free security audit logging for windows,” in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC 2015, 2015.

[24] S. Ma, X. Zhang, and D. Xu, “Protracer: Towards practical provenance tracing by alternating between logging and tainting,” in *NDSS*, 2016.

[25] Mozilla Developers Network, “Using images,” https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Using_images.

[26] C. Neasbitt, B. Li, R. Perdisci, L. Lu, K. Singh, and K. Li, “Webcapsule: Towards a lightweight forensic engine for web browsers,” in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’15, 2015.

[27] C. Neasbitt, R. Perdisci, K. Li, and T. Nelms, “Clickminer: Towards forensic reconstruction of user-browser interactions from network traces,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’14, 2014.

[28] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad, “Webwitness: Investigating, categorizing, and mitigating malware download paths,” in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC’15, 2015.

[29] ——, “Towards measuring and mitigating social engineering software download attacks,” in *Proceedings of the 25th USENIX Conference on Security Symposium*, ser. SEC’16, 2016.

[30] J. Palme, A. Hopmann, and N. Shelness, “Mime encapsulation of aggregate documents, such as html (mhtml),” 1999, https://tools.ietf.org/html/rfc2557.

[31] J. Saltzer and M. Schroeder, “The protection of information in computer systems,” http://web.mit.edu/Saltzer/www/publications/protection/.

[32] The Chromium Project, “Blick scheduler,” https://docs.google.com/document/d/16f_RIhZa47uEK_OdtTgzWdRU0RFMTQWMpEWyWXIpXUo/edit#heading=h.srz53flt1rrp.

[33] ——, “Compositor thread architecture,” https://www.chromium.org/developers/design-documents/compositor-thread-architecture.

[34] ——, “GPU accelerated compositing in Chrome,” https://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome.

[35] ——, “How chromium displays web pages,” https://www.chromium.org/developers/design-documents/displaying-a-web-page-in-chrome.

[36] ——, “Inter-process communication,” https://www.chromium.org/developers/design-documents/inter-process-communication.

[37] ——, “Multi-process architecture,” https://www.chromium.org/developers/design-documents/multi-process-architecture.

[38] ——, “Out-of-process iframes,” http://www.chromium.org/developers/design-documents/oop-iframes.

[39] ——, “Proposal for frame capture content API,” https://docs.google.com/document/d/1gRndVmVn7gWJ-rbIHaaOMNsCjSIBn4CAJbZuwLM2ROE/edit.

[40] ——, “The rendering critical path,” https://www.chromium.org/developers/the-rendering-critical-path.

[41] ——, "Scheduling js timer execution," https://docs.google.com/document/d/163ow-1wjd6L0rAN3V_U6t12eqVkq4mXDDjVaA4OuvCA/edit#.

[42] ——, "Threading," https://www.chromium.org/developers/design-documents/threading.

[43] K. Thomas, J. A. E. Crespo, R. Rasti, J.-M. Picod, C. Phillips, M.-A. Decoste, C. Sharp, F. Tirelo, A. Tofigh, M.-A. Courteau, L. Ballard, R. Shield, N. Jagpal, M. A. Rajab, P. Mavrommatis, N. Provos, E. Bursztein, and D. McCoy, "Investigating commercial pay-per-install and the distribution of unwanted software," in *25th USENIX Security Symposium (USENIX Security 16)*, Aug. 2016.

[44] N. Tolia, D. G. Andersen, and M. Satyanarayanan, "Quantifying interactive user experience on thin clients," *Computer*, vol. 39, no. 3, pp. 46–52, March 2006.

[45] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis." in *NDSS*, vol. 2007, 2007, p. 12.